CHAPTER 15

Machine Learning

This chapter explores machine learning. This topic is closely related to statistical modeling, which we considered in Chapter 14 because both use data to describe and predict outcomes of uncertain or unknown processes. The approach taken in statistical modeling emphasizes understanding how the data is generated by devising models that describe the underlying process behavior and fitting the model's parameters to the observed data. If the model fits the data well and satisfies the relevant model assumptions, then the model can be assumed to give a holistic description of the process. Such a model can, for example, be used to compute statistics with known distributions and evaluate statistical tests. However, if the data is too complex to be explained using available statistical models, this approach has reached its limits. In machine learning, on the other hand, the process that generates the data and potential models thereof is not central. Instead, the observed data and the explanatory variables are the fundamental starting point of a machine-learning application. Given data, machine-learning methods can be used to find patterns and structures in the data, which can be used to predict the outcome of new observations. Machine learning, therefore, does not provide an understanding of how data is generated. Because fewer assumptions are made regarding the distribution and statistical properties of the data, we typically cannot compute statistics and perform statistical tests regarding the significance of specific observations. Instead, machine learning strongly emphasizes the accuracy with which new observations are predicted.

Although significant differences exist in the fundamental approach taken in statistical modeling and machine learning, many mathematical methods are closely related or sometimes even the same. This chapter recognizes several methods used in Chapter 14 on statistical modeling, but they are employed with a different mindset and slightly different goals.

The chapter introduces basic machine-learning methods and surveys how such methods can be used in Python. The focus is on machine-learning methods that have broad application in many scientific and technical computing fields. The most prominent and comprehensive machine-learning library for Python is scikit-learn. However, there are several alternative and complementary libraries: TensorFlow, Keras, and PyTorch, to mention a few. This chapter uses the scikit-learn library exclusively, which implements the most common machine-learning algorithm. However, readers interested in machine learning are encouraged to explore the other libraries mentioned.

scikit-learn The scikit-learn library contains a comprehensive collection of machine-learning-related algorithms, including regression, classification, dimensionality reduction, and clustering. For more information about the project and its documentation, see its web page at `http://scikit-learn.org`. At the time of writing, the latest version of scikit-learn is 1.3.0.

R. Johansson, *Numerical Python*, https://doi.org/10.1007/979-8-8688-0413-7_15

Importing Modules

This chapter works with the scikit-learn library, which provides the `sklearn` Python module. With the `sklearn` module, we use the same import strategy as the SciPy library to explicitly import modules from the library. Let's use the following modules from the `sklearn` library.

```
In [1]: from sklearn import datasets
In [2]: from sklearn import model_selection
In [3]: from sklearn import linear_model
In [4]: from sklearn import metrics
In [5]: from sklearn import tree
In [6]: from sklearn import neighbors
In [7]: from sklearn import svm
In [8]: from sklearn import ensemble
In [9]: from sklearn import cluster
```

For plotting and basic numerical computation, we also require the Matplotlib and NumPy libraries, which we import in the usual manner.

```
In [10]: import matplotlib.pyplot as plt
In [11]: import numpy as np
```

We also use the Seaborn library for graphics and figure styling.

```
In [12]: import seaborn as sns
```

Brief Review of Machine Learning

Machine learning is a topic in the artificial intelligence field of computer science. Machine learning can include all applications where feeding training data into a computer program enables it to perform a given task. This is a very broad definition, but machine learning is often associated with a much more specific set of techniques and methods. Here, we take a practical approach and explore several basic methods and key concepts in machine learning by example. Let's begin with a brief introduction of the terminology and core concepts.

In machine learning, the process of fitting a model or an algorithm to observed data is known as *training*. Machine-learning applications can broadly be classified into either of two types: *supervised* and *unsupervised* learning, which differ in the type of data the application is trained with. In *supervised learning*, the data includes feature variables and known response variables. Both feature and response variables can be continuous or discrete. Preparing such data typically requires manual effort and sometimes even expert domain knowledge. The application is thus trained with handcrafted data, and the training can, therefore, be viewed as supervised machine learning. Examples of applications include regression (prediction of a continuous response variable) and classification (prediction of a discrete response variable), where the value of the response variable is known for the training dataset but not for new samples.

In contrast, *unsupervised learning* corresponds to situations where machine-learning applications are trained with raw data that is not labeled or otherwise manually prepared. An example of unsupervised learning is the clustering of data into groups, or in other words, the grouping of data into initially unknown categories. In contrast to supervised classification, it is typical for unsupervised learning that the final categories are not known in advance, and the training data, therefore, cannot be labeled accordingly. It may also be the case that the manual labeling of the data is difficult or costly, for example, because the number of samples is too large. Unsupervised machine learning is more complicated and limited in what it can be used for than supervised machine learning, which should be preferred whenever possible. However, unsupervised machine learning can be a powerful tool when creating labeled training datasets is impossible or unrealistic.

Naturally, there is much more complexity to machine learning than suggested by the basic types of problems outlined in the preceding text. But these concepts are recurring themes in many machine-learning applications. This chapter looks at a few basic machine-learning techniques demonstrating several central concepts. First, let's go over some common machinelearning terminology.

- *Cross-validation* is dividing the available data into *training data* and *testing data* (also known as *validation data*), where only the training data is used to train the machine-learning model and where the test data allows the trained application to be tested on previously unseen data. This aims to measure how well the model predicts new observations and limit problems with overfitting. There are several approaches to dividing the data into training and testing datasets. For example, one extreme approach is to test all possible ways to divide the data (*exhaustive cross-validation*) and use an aggregate result (e.g., average or the minimum value, depending on the situation). However, for large datasets, the number of possible training and testing data combinations becomes extremely large, making exhaustive cross-validation impractical. Another extreme is to use all but one sample in the training set and the remaining sample in the training set (leave-one-out cross-validation) and to repeat the training-test cycle for all combinations in which one sample is chosen from the available data. A variant of this method is to divide the available data into *k* groups and perform a leave-one-out cross-validation with the *k* groups of datasets. This method is known as *k*-fold cross-validation and is a popular technique often used in practice. In the scikit-learn library, the `sklearn.model_selection` module contains functions for working with cross-validation.
- *Feature extraction* is an important step in the preprocessing stage of a machine-learning problem. It involves creating suitable feature variables and the corresponding feature matrices that can be passed to one of many machine-learning algorithms implemented in the scikit-learn library. The scikit-learn `sklearn.feature_extraction` module plays a similar role in many machine-learning applications as the Patsy formula library does in statistical modeling, especially for text- and image-based machine-learning problems. Using methods from the `sklearn.feature_extraction` module, we can automatically assemble feature matrices (design matrices) from various data sources.
- *Dimensionality reduction* and *feature selection* are techniques frequently used in machine-learning applications where it is common to have a large number of explanatory variables (features), many of which may not significantly contribute to the predictive power of the application. To reduce the complexity of the model, it is often desirable to eliminate less useful features, thereby reducing the problem's dimensionality. This is particularly important when the number of features is comparable to or larger than the number of observations. The scikit-learn `sklearn.decomposition` and `sklearn.feature_selection` modules contain functions for reducing the dimensionality of a machine-learning problem: for example, principal component analysis (PCA) is a popular technique for dimensionality reduction that works by performing a singular-value decomposition of the feature matrix and keeping only dimensions that correspond to the most significant singular vectors.

The following sections look at how scikit-learn can be used to solve examples of machine-learning problems using the techniques discussed in the preceding text. Here, we work with generated data and built-in datasets. Like the statsmodels library, scikit-learn comes with several built-in datasets to explore machine-learning methods. The `datasets` module in `sklearn` provides three groups of functions for loading built-in datasets (with the `load_` prefix; e.g., `load_wine`), fetching external datasets (with the `fetch_` prefix; e.g., `fetch_californa_housing`), and generating datasets from random numbers (with the `make_` prefix; e.g., `make_regression`).

Regression

Regression is central to machine learning and statistical modeling, as demonstrated in Chapter 14. In machine learning, we are not primarily concerned with how well the regression model fits the data but instead care about how well it predicts new observations. For example, suppose we have a large number of features and a smaller number of observations. In that case, we can often fit the regression perfectly to the data without it being beneficial for predicting new values. This is an example of overfitting: A small residual between the data and the regression model does not guarantee that the model can accurately predict future observations. In machine learning, a common method to deal with this problem is to partition the available data into a training and testing datasets for validating the regression results against previously unseen data.

To see how fitting a training dataset and validating the result against a testing dataset can work out, consider a regression problem with 50 samples and 50 features, out of which only 10 features are informative (linearly correlated with the response variable). This simulates a scenario when we have 50 known features, but it turns out that only 10 of those features contribute to the predictive power of the regression model. The `make_regression` function in the `sklearn.datasets` module generates data of kind.

```
In [13]: X_all, y_all = datasets.make_regression(
    ...:     n_samples=50, n_features=50, n_informative=10)
```

The result is two arrays, `X_all` and `y_all`, of shapes `(50, 50)` and `(50,)`, corresponding to the design matrices for a regression problem with 50 samples and 50 features. Instead of performing a regression on the entire dataset (and obtaining a perfect fit because of the small number of observations), we split the dataset into two equal-sized datasets using the `train_test_split` function from `sklearn.model_selection` module. The result is a training dataset `X_train`, `y_train`, and a testing dataset `X_test`, `y_test`.

```
In [14]: X_train, X_test, y_train, y_test = \
    ...:     model_selection.train_test_split(X_all, y_all, train_size=0.5)
```

In scikit-learn, ordinary linear regression can be carried out using the `LinearRegression` class from the `sklearn.linear_model` module, which is comparable with the `statsmodels.api.OLS` from the statsmodels library. To perform a regression, we first create a `LinearRegression` instance.

```
In [15]: model = linear_model.LinearRegression()
```

To fit the model to the data, we must invoke the `fit` method, which takes the feature matrix and the response variable vector as the first and second arguments.

```
In [16]: model.fit(X_train, y_train)
Out[16]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
                          normalize=False)
```

Note that compared to the `OLS` class in statsmodels, the order of the feature matrix and response variable vector is reversed, and in statsmodels the data is specified when the class instance is created instead of when calling the `fit` method. Also, in scikit-learn, calling the `fit` method does not return a new result object; instead, the result is stored directly in the model instance. These minor differences are small inconveniences when working interchangeably with the statsmodels and scikit-learn modules and are worth noting.[1]

Since the regression problem has 50 features and we only trained the model with 25 samples, we can expect complete overfitting that perfectly fits the data. This can be quantified by computing the sum of squared errors (SSEs) between the model and the data. To evaluate the model for a given set of features, we can use the `predict` method to compute the residuals and the SSE.

```
In [17]: def sse(resid):
    ...:     return np.sum(resid**2)
In [18]: resid_train = y_train - model.predict(X_train)
    ...: sse_train = sse(resid_train)
    ...: sse_train
Out[18]: 8.1172209425431673e-25
```

As expected, the residuals are all essentially zero for the training dataset due to the overfitting allowed by having twice as many features as data points. This overfitted model is, however, not at all suitable for predicting unseen data. This can be verified by computing the SSE for our test dataset.

```
In [19]: resid_test = y_test - model.predict(X_test)
    ...: sse_test = sse(resid_test)
    ...: sse_test
Out[19]: 213555.61203039082
```

The result is a very large SSE value, which indicates that the model does not predict new observations well. An alternative measure of the fit of a model to a dataset is the R-squared score (see Chapter 14), which we can compute using the `score` method. It takes a feature matrix and response variable vector as arguments and calculates the score. As expected, we obtain an r-square score of 1.0 for the training dataset, but for the testing dataset, we get a low score.

```
In [20]: model.score(X_train, y_train)
Out[20]: 1.0
In [21]: model.score(X_test, y_test)
Out[21]: 0.31407400675201746
```

The big difference between the training and testing datasets scores indicates that the model is overfitted.

Finally, we can also take a graphical approach and plot the residuals of the training and testing datasets and visually inspect the values of the coefficients and the residuals. From a `LinearRegression` object, we can extract the fitted parameters using the `coef_` attribute. To simplify repeated plotting of the training and testing residuals and the model parameters, we first create a `plot_residuals_and_coeff` function for plotting these quantities. We then call the function with the result from the ordinary linear regression model trained and tested on the training and testing datasets, respectively. The result is shown in Figure 15-1, and there is a significant difference in the magnitude of the residuals for the test and the training datasets for every sample.

[1] In practice it is common to work with both statsmodels and scikit-learn, as they in many respects complement each other. However, this chapter focuses solely on scikit-learn.

```
In [22]: def plot_residuals_and_coeff(resid_train, resid_test, coeff):
    ...:     fig, axes = plt.subplots(1, 3, figsize=(12, 3))
    ...:     axes[0].bar(np.arange(len(resid_train)), resid_train)
    ...:     axes[0].set_xlabel("sample number")
    ...:     axes[0].set_ylabel("residual")
    ...:     axes[0].set_title("training data")
    ...:     axes[1].bar(np.arange(len(resid_test)), resid_test)
    ...:     axes[1].set_xlabel("sample number")
    ...:     axes[1].set_ylabel("residual")
    ...:     axes[1].set_title("testing data")
    ...:     axes[2].bar(np.arange(len(coeff)), coeff)
    ...:     axes[2].set_xlabel("coefficient number")
    ...:     axes[2].set_ylabel("coefficient")
    ...:     fig.tight_layout()
    ...:     return fig, axes
In [23]: fig, ax = plot_residuals_and_coeff(resid_train, resid_test, model.coef_)
```

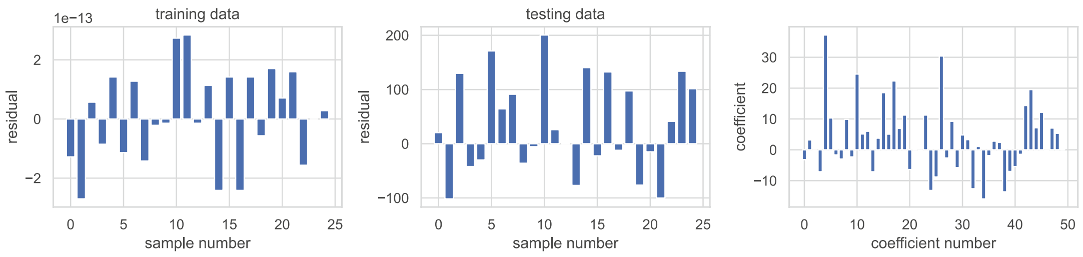

***Figure 15-1.** The residual between the ordinary linear regression model and the training data (left), the model and the test data (middle), and the values of the coefficients for the 50 features (right)*

Overfitting in this example happens because we have too few samples, and one solution could be to collect more samples until overfitting is no longer a problem. However, this may not always be practical, as collecting observations may be expensive, and because in some applications, we might have a very large number of features. For such situations, it is desirable to be able to fit a regression problem in a way that avoids overfitting as much as possible (at the expense of not fitting the training data perfectly) so that the model can give meaningful predictions for new observations.

Regularized regression is one possible solution to this problem. The following looks at a few variations of regularized regression. In ordinary linear regression, the model parameters are chosen to minimize the sum of squared residuals. Viewed as an optimization problem, the objective function is, therefore, $\min_\beta \|X\beta - y\|_2^2$, where X is the feature matrix, y is the response variables, and β is the vector of model parameters and where $\|\cdot\|_2$ denotes the L2 norm. In *regularized* regression, we add a *penalty term* in the objective function of the minimization problem. Different types of penalty terms impose different types of regularization on the original regression problem. Two popular regularizations are obtained by adding the L1 or L2 norms of the parameter vector to the minimization objective function, $\min_\beta \left\{\|X\beta - y\|_2^2 + \alpha\|\beta\|_1\right\}$ and $\min_\beta \left\{\|X\beta - y\|_2^2 + \alpha\|\beta\|_2^2\right\}$. These are known as LASSO and Ridge regression, respectively. Here, α is a free parameter that determines the strength of the regularization. Adding the L2 norm $\|\beta\|_2^2$ favors model parameter vectors with smaller coefficients, and adding the L1 norm $\|\beta\|_1$ favors a model parameter vectors with as few nonzero elements as possible. Which type of regularization is more suitable depends on the problem at hand: When we wish to eliminate as many features as possible, we can use L1 regularization with LASSO regression, and when we want to limit the magnitude of the model coefficients, we can use L2 regularization with Ridge regression.

With scikit-learn, we can perform Ridge regression using the `Ridge` class from the `sklearn.linear_model` module. The usage of this class is almost the same as the `LinearRegression` class used in the preceding text, but we can also give the value of the α parameter that determines the strength of the regularization as an argument when we initialize the class. Here, we chose the value $\alpha = 2.5$. A more systematic approach to choosing α is introduced later in this chapter.

```
In [24]: model = linear_model.Ridge(alpha=2.5)
```

To fit the regression model to the data, we again use the fit method, passing the training feature matrix and response variable as arguments.

```
In [25]: model.fit(X_train, y_train)
Out[25]: Ridge(alpha=2.5, copy_X=True, fit_intercept=True, max_iter=None,
               normalize=False, solver='auto', tol=0.001)
```

Once the model has been fitted to the training data, we can compute the model predictions for the training and testing datasets and compute the corresponding SSE values.

```
In [26]: resid_train = y_train - model.predict(X_train)
    ...: sse_train = sse(resid_train)
    ...: sse_train
Out[26]: 178.50695164950841
In [27]: resid_test = y_test - model.predict(X_test)
    ...: sse_test = sse(resid_test)
    ...: sse_test
Out[27]: 212737.00160105844
```

We note that the SSE of the training data is no longer close to zero, but there is a slight decrease in the SSE for the testing data. For comparison with ordinary regression, we also plot the training and testing residuals and the model parameters using the `plot_residuals_and_coeff` function defined in the preceding text. The result is shown in Figure 15-2.

```
In [28]: fig, ax = plot_residuals_and_coeff(resid_train, resid_test, model.coef_)
```

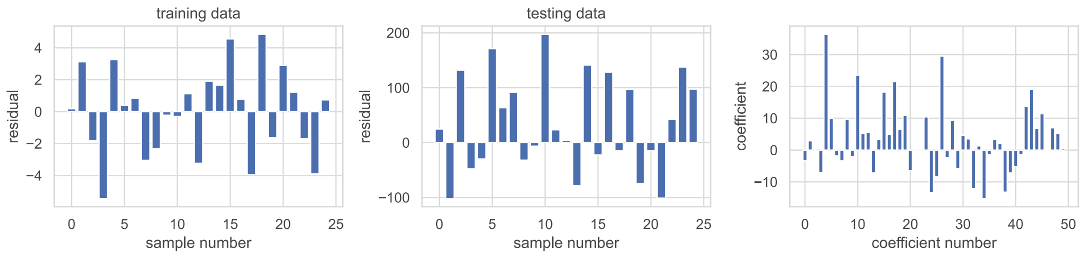

Figure 15-2. *The residual between the Ridge-regularized regression model and the training data (left), the model and the test data (middle), and the values of the coefficients for the 50 features (right)*

Similarly, we can perform the L1-regularized LASSO regression using the `Lasso` class from the `sklearn.linear_model` module. It also accepts the value of the α parameter as an argument when the class instance is initialized. Here, we choose $\alpha = 1.0$ and perform the model fitting to the training data and the computation of the SSE for the training and testing data in the same way described previously.

```
In [29]: model = linear_model.Lasso(alpha=1.0)
In [30]: model.fit(X_train, y_train)
Out[30]: Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
               normalize=False, positive=False, precompute=False,
               random_state=None, selection='cyclic', tol=0.0001,
               warm_start=False)
In [31]: resid_train = y_train - model.predict(X_train)
    ...: sse_train = sse(resid_train)
    ...: sse_train
Out[31]: 309.74971389531891
In [32]: resid_test = y_test - model.predict(X_test)
    ...: sse_test = sse(resid_test)
    ...: sse_test
Out[32]: 1489.1176065002333
```

Here, we note that while the SSE of the training data increased compared to that of the ordinary regression, the SSE for the testing data decreased significantly. Thus, by paying a price for how well the regression model fits the training data, we have obtained a model with a significantly improved ability to predict the testing dataset. For comparison with the earlier methods, we again graph the residuals and the model parameters with the `plot_residuals_and_coeff` function. The result is shown in Figure 15-3. In the rightmost panel of this figure, we see that the coefficient profile is significantly different from those shown in Figure 15-1 and Figure 15-2, and the coefficient vector produced with the LASSO regression contains mostly zeros. This is a suitable method for the current data because, in the beginning, when we generated the dataset, we chose 50 features, out of which only 10 are informative. If we suspect we might have many features that might not contribute much to the regression model, using the L1 regularization of the LASSO regression can thus be a good approach to try.

```
In [33]: fig, ax = plot_residuals_and_coeff(resid_train, resid_test, model.coef_)
```

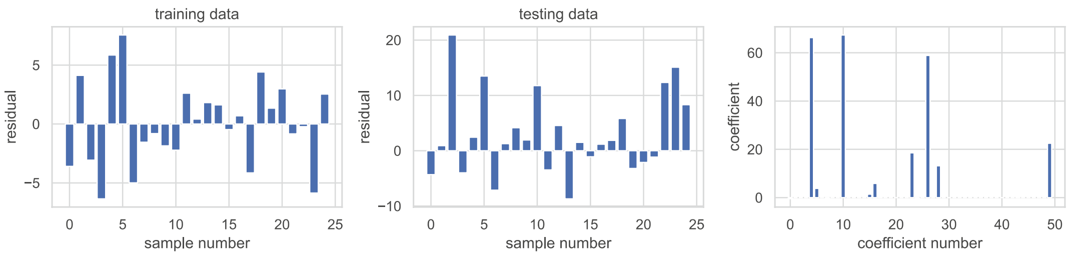

Figure 15-3. *The residual between the LASSO-regularized regression model and the training data (left), the model and the test data (middle), and the values of the coefficients for the 50 features (right)*

The values of α used in the two previous examples using Ridge and LASSO regression were chosen arbitrarily. The most suitable value of α is problem-dependent, and for every new problem, we need to find an appropriate value using trial and error. The scikit-learn library provides methods for assisting this process. But before we explore those methods, it is instructive to look at how the regression model parameters and the SSE for the training and testing datasets depend on the value of α for a specific problem. Here, we focus on LASSO regression since it was seen to work well for the current problem, and we repeatedly solve the same problem using different values for the regularization strength parameter α while storing the values of the coefficients and SSE values in NumPy arrays.

Let's begin by creating the required NumPy arrays. We use `np.logspace` to create a range of α values that span several orders of magnitude.

```
In [34]: alphas = np.logspace(-4, 2, 100)
In [35]: coeffs = np.zeros((len(alphas), X_train.shape[1]))
In [36]: sse_train = np.zeros_like(alphas)
In [37]: sse_test = np.zeros_like(alphas)
```

Next, we loop through the α values and perform the LASSO regression for each value.

```
In [38]: for n, alpha in enumerate(alphas):
    ...:     model = linear_model.Lasso(alpha=alpha)
    ...:     model.fit(X_train, y_train)
    ...:     coeffs[n, :] = model.coef_
    ...:     sse_train[n] = sse(y_train - model.predict(X_train))
    ...:     sse_test[n] = sse(y_test - model.predict(X_test))
```

Finally, we plot the coefficients and the SSE for the training and testing datasets using Matplotlib. The result is shown in Figure 15-4. We can see in the left panel of this figure that many coefficients are nonzero for very small values of α. This corresponds to the overfitting regime. We can also see that when α is increased above a certain threshold, many of the coefficients collapse to zero, and only a few coefficients remain nonzero. In the right panel of the figure, we see that while the SSE for the training set is steadily increasing with increasing α, there is also a sharp drop in the SSE for the testing dataset. This is the sought-after effect in LASSO regression. However, for too large values of α, all coefficients converge to zero, and the SSEs for both the training and testing datasets become large. Therefore, there is an optimal region of α that prevents overfitting and improves the model's ability to predict unseen data. While these observations are not universally true, a similar pattern can be seen for many problems.

```
In [39]: fig, axes = plt.subplots(1, 2, figsize=(12, 4), sharex=True)
    ...: for n in range(coeffs.shape[1]):
    ...:     axes[0].plot(np.log10(alphas), coeffs[:, n], color='k', lw=0.5)
    ...:
    ...: axes[1].semilogy(np.log10(alphas), sse_train, label="train")
    ...: axes[1].semilogy(np.log10(alphas), sse_test, label="test")
    ...: axes[1].legend(loc=0)
    ...:
    ...: axes[0].set_xlabel(r"${\log_{10}}\alpha$", fontsize=18)
    ...: axes[0].set_ylabel(r"coefficients", fontsize=18)
    ...: axes[1].set_xlabel(r"${\log_{10}}\alpha$", fontsize=18)
    ...: axes[1].set_ylabel(r"sse", fontsize=18)
```

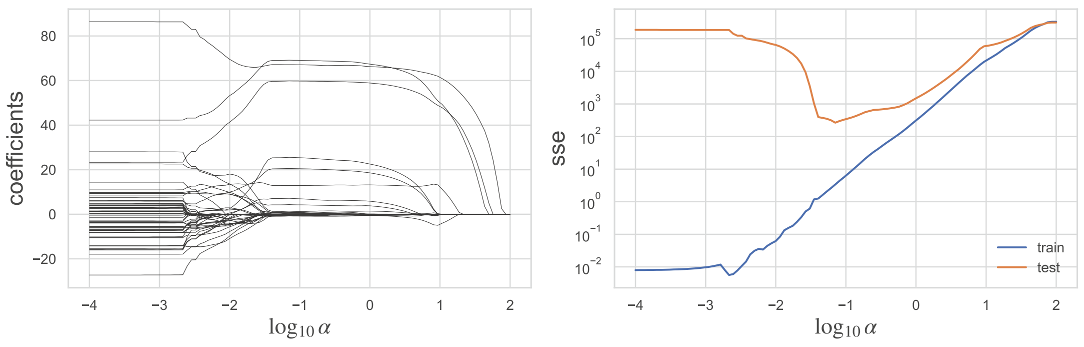

Figure 15-4. *The coefficients (left) and the sum of squared errors (SSEs) for the training and testing datasets (right), for LASSO regression as a function of the logarithm of the regularization strength parameter α*

Testing a regularized regression with several values of α can be carried out automatically using, for example, the `RidgeCV` and `LassoCV` classes. These Ridge and LASSO regression variants internally search for the optimal α using a cross-validation approach. By default, a k-fold cross-validation with $k = 3$ is used, although this can be changed using the `cv` argument to these classes. Because of the built-in cross-validation, we do not need to explicitly divide the dataset into training and testing datasets, as we have done previously.

To use the LASSO method with an automatically chosen α, we create an instance of `LassoCV` and invoke its `fit` method.

```
In [40]: model = linear_model.LassoCV()
In [41]: model.fit(X_all, y_all)
Out[41]: LassoCV(alphas=None, copy_X=True, cv=None, eps=0.001, fit_intercept=True,
                max_iter=1000, n_alphas=100, n_jobs=1, normalize=False,
                positive=False, precompute='auto', random_state=None,
                selection='cyclic', tol=0.0001, verbose=False)
```

The value of regularization strength parameter α selected through the crossvalidation search is accessible through the `alpha_` attribute.

```
In [42]: model.alpha_
Out[42]: 0.13118477495069433
```

The suggested value of α agrees reasonably well with what we might have guessed from Figure 15-4. For comparison with the previous method, we also compute the SSE for the training and testing datasets (although both were used for training in the call to `LassoCV.fit`) and graph the SSE values together with the model parameters, as shown in Figure 15-5. Using the cross-validated LASSO method obtains a model that predicts both the training and testing datasets with relatively high accuracy, and we are no longer as likely to suffer from the problem of overfitting despite having few samples compared to the number of features.[2]

```
In [43]: resid_train = y_train - model.predict(X_train)
    ...: sse_train = sse(resid_train)
    ...: sse_train
```

[2] However, note that we can never be sure that a machine-learning application does not suffer from overfitting before we see how the application performs on new observations, and a repeated reevaluation of the application on a regular basis is a good practice.

```
Out[43]: 66.900068715063625
In [44]: resid_test = y_test - model.predict(X_test)
    ...: sse_test = sse(resid_test)
    ...: sse_test
Out[44]: 966.39293785448456
In [45]: fig, ax = plot_residuals_and_coeff(resid_train, resid_test, model.coef_)
```

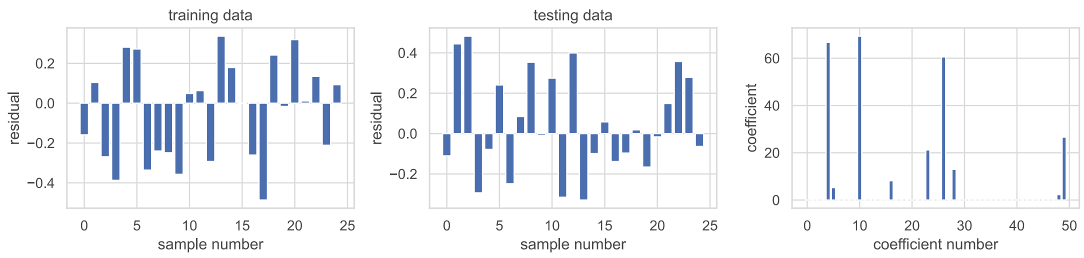

***Figure 15-5.** The residuals of the LASSO-regularized regression model with cross-validation for the training data (left) and the testing data (middle). The values of the coefficients for the 50 features are also shown (right)*

Finally, another type of popular regularized regression, which combines the L1 and L2 regularization of the LASSO and Ridge methods, is called elastic-net regularization. The minimization objective function for this method is $\min_\beta \left\{ \| X\beta - y \|_2^2 + \alpha\rho \| \beta \|_1 + \alpha(1-\rho) \| \beta \|_2^2 \right\}$, where the parameter ρ (`l1_ratio` in scikit-learn) determines the relative weight of the L1 and L2 penalties and, thus, how much the method behaves like the LASSO and Ridge methods. In scikit-learn, we can perform an elastic-net regression using the `ElasticNet` class, to which we can give explicit values of the α (`alpha`) and ρ (`l1_ratio`) parameters, or the cross-validated version `ElasticNetCV`, which automatically finds suitable values of the α and ρ parameters.

```
In [46]: model = linear_model.ElasticNetCV()
In [47]: model.fit(X_train, y_train)
Out[47]: ElasticNetCV(alphas=None, copy_X=True, cv=None, eps=0.001,
                      fit_intercept=True, l1_ratio=0.5, max_iter=1000,
                      n_alphas=100, n_jobs=1, normalize=False, positive=False,
                      precompute='auto', random_state=None, selection='cyclic',
                      tol=0.0001, verbose=0)
```

The value of regularization parameters α and ρ suggested by the crossvalidation search is available through the `alpha_` and `l1_ratio` attributes.

```
In [48]: model.alpha_
Out[48]: 0.13118477495069433
In [49]: model.l1_ratio
Out[49]: 0.5
```

For comparison with the previous method, we again compute the SSE and plot the model coefficients, as shown in Figure 15-6. As expected with $\rho = 0.5$, the result has characteristics of both LASSO regression (favoring a sparse solution vector with only a few dominating elements) and Ridge regression (suppressing the magnitude of the coefficients).

```
In [50]: resid_train = y_train - model.predict(X_train)
    ...: sse_train = sse(resid_train)
    ...: sse_train
Out[50]: 2183.8391729391255
In [51]: resid_test = y_test - model.predict(X_test)
    ...: sse_test = sse(resid_test)
    ...: sse_test
Out[51]: 2650.0504463382508
In [52]: fig, ax = plot_residuals_and_coeff(resid_train, resid_test, model.coef_)
```

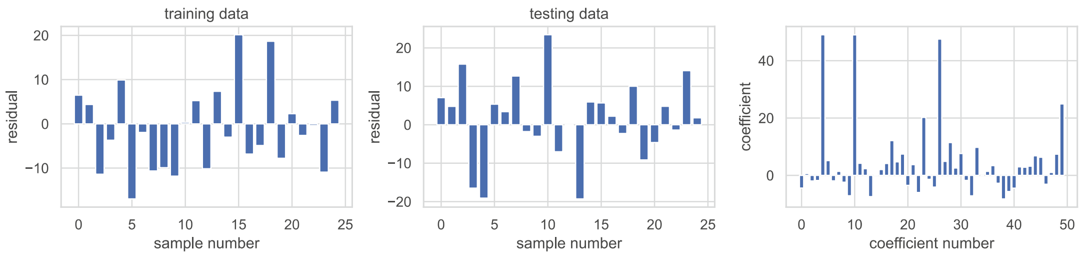

***Figure 15-6.** The residuals of the elastic-net regularized regression model with cross-validation for the training data (left) and the testing data (middle). The values of the coefficients for the 50 features are also shown (right)*

Classification

Like regression, classification is a central topic in machine learning. The chapter used a logistic regression model to classify observations into discrete categories. Logistic regression is also used in machine learning for the same task. But there is also a wide variety of alternative algorithms for classification, such as nearest neighbor methods, support vector machines (SVM), decision trees, and random forest methods. The scikit-learn library provides a convenient unified API that allows all these different methods to be used interchangeably for any classification problem.

To see how we can train a classification model with a training dataset and test its performance on testing datasets, let's once again look at the Iris dataset, which provides features for Iris flower samples (sepal and petal width and height), together with the species of each sample (*setosa*, *versicolor*, and *virginica*). The Iris dataset included in the scikit-learn library (as well as in the statsmodels library) is a classic dataset commonly used for testing and demonstrating machine-learning algorithms and statistical models. Here, we revisit the problem of classifying the species of a flower sample given its sepal and petal width and height (see also Chapter 14). First, call the `load_iris` function in the dataset module to load the dataset. The result is a container object (called a `Bunch` object in the scikit-learn jargon) containing the data and metadata.

```
In [53]: iris = datasets.load_iris()
In [54]: type(iris)
Out[54]: sklearn.utils._bunch.Bunch
```

For example, descriptive names of the features and target classes are available through the `feature_names` and `target_names` attributes.

```
In [55]: iris.target_names
Out[55]: array(['setosa', 'versicolor', 'virginica'], dtype='|S10')
In [56]: iris.feature_names
Out[56]: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

The actual dataset is available through the `data` and `target` attributes.

```
In [57]: iris.data.shape
Out[57]: (150, 4)
In [58]: iris.target.shape
Out[58]: (150,)
```

We begin by splitting the dataset into a training and testing part, using the `train_test_split` function. Here, we include 70% of the samples in the training set, leaving the remaining 30% for testing and validation.

```
In [59]: X_train, X_test, y_train, y_test = \
    ...:     model_selection.train_test_split(
    ...:         iris.data, iris.target, train_size=0.7)
```

The first step in training a classifier and performing classification tasks using scikitlearn is to create a classifier instance. As mentioned, numerous classifiers are available. Let's begin with a logistic regression classifier, which is provided by the `LogisticRegression` class in the `linear_model` module.

```
In [60]: classifier = linear_model.LogisticRegression()
```

The classifier's training is carried out by calling the `fit` method of the classifier instance. The arguments are the design matrices for the feature and target variables. Here, we use the training part of the Iris dataset arrays created for us when loading the dataset using the `load_iris` function. If the design matrices are not already available, we can use the same techniques used in Chapter 14, such as constructing the matrices by hand using NumPy functions or using the Patsy library to construct the appropriate arrays automatically. We can also use the feature extraction utilities in the `feature_extraction` module in the scikit-learn library.

```
In [61]: classifier.fit(X_train, y_train)
Out[61]: LogisticRegression(C=1.0, class_weight=None, dual=False,
                            fit_intercept=True, intercept_scaling=1, max_iter=100,
                            multi_class='ovr', penalty='l2', random_state=None,
                            solver='liblinear', tol=0.0001, verbose=0)
```

Once the classifier has been trained, we can immediately start using it to predict the class for new observations using the `predict` method. Here, we apply this method to predict the class for the samples assigned to the testing dataset to compare the predictions with the actual values.

```
In [62]: y_test_pred = classifier.predict(X_test)
```

The sklearn.metrics module contains helper functions to assist in analyzing the performance and accuracy of classifiers. For example, the classification_report function, which takes arrays of actual values and the predicted values, returns a tabular summary of the informative classification metrics related to the rate of false negatives and false positives[3].

```
In [63]: print(metrics.classification_report(y_test, y_test_pred))
             precision    recall  f1-score   support
          0       1.00      1.00      1.00        13
          1       1.00      0.92      0.96        13
          2       0.95      1.00      0.97        19
avg / total       0.98      0.98      0.98        45
```

The so-called *confusion matrix*, which can be computed using the confusion_matrix function, also presents useful classification metrics in a compact form: The diagonals correspond to the number of samples that are correctly classified for each level of the category variable, and the off-diagonal elements are the number of incorrectly classified samples. More specifically, the element C_{ij} of the confusion matrix C is the number of samples of category i that were categorized as j. For the current data, we obtain the confusion matrix.

```
In [64]: metrics.confusion_matrix(y_test, y_test_pred)
Out[64]: array([[13  0  0]
                [ 0 12  1]
                [ 0  0 19]])
```

This confusion matrix shows that all elements in the first and third classes were classified correctly, but one element of the second class was mistakenly classified as class three. Note that the elements in each row of the confusion matrix sum up the total number of samples for the corresponding category. In this testing sample, we have 13 elements each in the first and second class and 19 elements in the third class, as can be seen by counting unique values in the y_test array.

```
In [65]: np.bincount(y_test)
Out[65]: array([13, 13, 19])
```

To perform a classification using a different classifier algorithm, we only need to create an instance of the corresponding classifier class. For example, to use a decision tree instead of logistic regression, we can use the DesicisionTreeClassifier class from the sklearn.tree module. Training the classifier and predicting new observations is done in the same way for all classifiers.

```
In [66]: classifier = tree.DecisionTreeClassifier()
    ...: classifier.fit(X_train, y_train)
    ...: y_test_pred = classifier.predict(X_test)
    ...: metrics.confusion_matrix(y_test, y_test_pred)
Out[66]: array([[13,  0,  0],
                [ 0, 12,  1],
                [ 0,  1, 18]])
```

The resulting confusion matrix with the decision tree classifier is somewhat different, corresponding to one additional misclassification in the testing dataset.

[3] Note that we do not get the same results each time we run this process due to the randomness in the train-test data split when using the model_selection.train_test_split function. If reproducibility is required, we can use the random_state keyword argument to ensure it.

Other popular classifiers available in scikit-learn include the nearest neighbor classifier `KNeighborsClassifier` from the `sklearn.neighbors` module, the support vector classifier (`SVC`) from the `sklearn.svm` module, and the random forest classifier `RandomForestClassifier` from the `sklearn.ensemble` module. Since they all have the same usage pattern, we can programmatically apply a series of classifiers on the same problem and compare their performance (on this particular problem), for example, as a function of the training and testing sample sizes. To this end, we create a NumPy array with training size ratios ranging from 10% to 90%.

```
In [67]: train_size_vec = np.linspace(0.1, 0.9, 30)
```

Next, we create a list of classifier classes to apply.

```
In [68]: classifiers = [tree.DecisionTreeClassifier,
    ...:                neighbors.KNeighborsClassifier,
    ...:                svm.SVC,
    ...:                ensemble.RandomForestClassifier]
```

We also create an array in which we can store the diagonals of the confusion matrix as a function of training size ratio and classifier.

```
In [69]: cm_diags = np.zeros((3, len(train_size_vec), len(classifiers)), dtype=float)
```

Finally, we loop over each training size ratio and classifier. For each combination, we train the classifier, predict the values of the testing data, compute the confusion matrix, and store its diagonal divided by the ideal values in the `cm_diags` array.

```
In [70]: for n, train_size in enumerate(train_size_vec):
    ...:     X_train, X_test, y_train, y_test = \
    ...:         model_selection.train_test_split(iris.data, iris.target,
    ...:                                          train_size=train_size)
    ...:     for m, Classifier in enumerate(classifiers):
    ...:         classifier = Classifier()
    ...:         classifier.fit(X_train, y_train)
    ...:         y_test_p = classifier.predict(X_test)
    ...:         cm_diags[:, n, m] = metrics.confusion_matrix(
    ...:             y_test, y_test_p).diagonal()
    ...:         cm_diags[:, n, m] /= np.bincount(y_test)
```

The resulting classification accuracy for each classifier, as a function of training size ratio, is plotted and shown in Figure 15-7.

```
In [71]: fig, axes = plt.subplots(1, len(classifiers), figsize=(12, 3))
    ...: for m, Classifier in enumerate(classifiers):
    ...:      axes[m].plot(train_size_vec, cm_diags[2, :, m],
    ...:                   label=iris.target_names[2])
    ...:      axes[m].plot(train_size_vec, cm_diags[1, :, m],
    ...:                   label=iris.target_names[1])
    ...:      axes[m].plot(train_size_vec, cm_diags[0, :, m],
    ...:                   label=iris.target_names[0])
    ...:      axes[m].set_title(type(Classifier()).__name__)
    ...:     axes[m].set_ylim(0, 1.1)
    ...:     axes[m].set_ylabel("classification accuracy")
```

```
   ...:     axes[m].set_xlabel("training size ratio")
   ...:     axes[m].legend(loc=4)
```

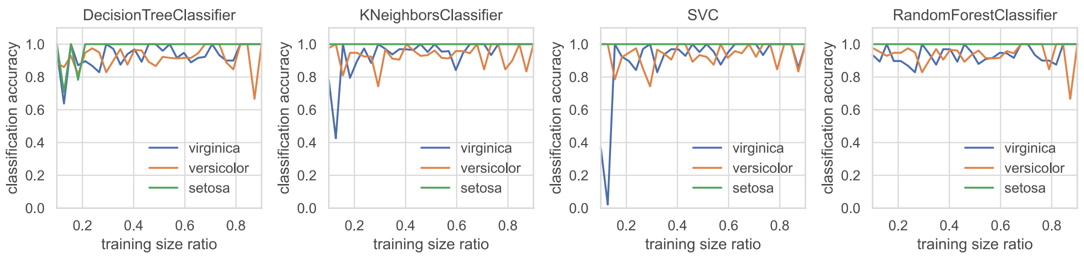

***Figure 15-7.** Comparison of classification accuracy of four different classifiers*

Figure 15-7 shows that the classification error is different in each model, but for this example, they have comparable performance. Which classifier is the best depends on the problem at hand, and it is difficult to give any definite answer to which one is more suitable in general. Fortunately, it is easy to switch between different classifiers in scikit-learn and, therefore, effortless to try a few different classifiers for a given classification problem. In addition to the classification accuracy, another important aspect is the computational performance and scaling to larger problems. For large classification problems with many features, decision-tree methods such as the randomized forest method are often a good starting point.

Clustering

In the two previous sections, we explored regression and classification, both examples of supervised learning, since the response variables are given in the dataset. Clustering is a different type of problem and an important topic in machine learning. It can be considered a classification problem where the classes are unknown, making clustering an example of unsupervised learning. The training dataset for a clustering algorithm contains only the feature variables, and the algorithm's output is an array of integers that assign each sample to a cluster (or class). This output array corresponds to the response variable in a supervised classification problem.

The scikit-learn library implements a large number of clustering algorithms that are suitable for different types of clustering problems and different types of datasets. Popular general-purpose clustering methods include the *K-means algorithm*, which groups the samples into clusters such that the within-group sum of square deviation from the group center is minimized, and the *mean-shift algorithm*, which clusters the samples by fitting the data to density functions (e.g., Gaussian functions).

In scikit-learn, the `sklearn.cluster` module contains several clustering algorithms, including the K-means algorithm `KMeans` and the mean-shift algorithm `MeanShift`, just to mention a few. To perform a clustering task with one of these methods, we first initialize an instance of the corresponding class and train it with a feature-only dataset using the `fit` method, and we finally obtain the result of the clustering by calling the `predict` method. Many clustering algorithms require the number of clusters as an input parameter, which we can specify using the `n_clusters` parameter when the class instance is created.

To demonstrate clustering, let's again consider the Iris dataset. Here, we do not use the response variable, which was used in supervised classification. Instead, we attempt to automatically discover a suitable clustering of the samples using the *K*-means method. We begin by loading the Iris data, as before, and store the feature and target data in the variables `X` and `y`, respectively.

```
In [72]: X, y = iris.data, iris.target
```

With the K-means clustering method, we must specify how many clusters we want in the output. The most suitable number of clusters is not always apparent in advance, and trying clustering with a few different numbers of clusters is often necessary. However, here, we know that the data corresponds to three species of Iris flowers, so we use three clusters. To perform the clustering, we create an instance of the `Kmeans` class, using the `n_clusters` argument to set the number of clusters.

```
In [73]: n_clusters = 3
In [74]: clustering = cluster.KMeans(n_clusters=n_clusters)
```

To perform the computation, we call the `fit` method with the Iris feature matrix as an argument.

```
In [75]: clustering.fit(X)
Out[75]: KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=3,
                n_init=10, n_jobs=1, precompute_distances='auto',
                random_state=None, tol=0.0001, verbose=0)
```

The clustering result is available through the `predict` method, to which we also pass a feature dataset that optionally can contain features of new samples. However, not all the clustering methods implemented in scikit-learn support predicting clusters for a new sample. In this case, the `predict` method is unavailable, and we need to use the `fit_predict` method instead. Here, we use the `predict` method with the training feature dataset to obtain the clustering result.

```
In [76]: y_pred = clustering.predict(X)
```

The result is an integer array of the same length and the number of samples in the training dataset. The elements in the array indicate which group (from `0` up to `n_samples-1`) each sample is assigned to. Since the resulting array `y_pred` is long, we only display every eighth element in the array using the NumPy stride indexing `::8`.

```
In [77]: y_pred[::8]
Out[77]: array([1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0],
               dtype=int32)
```

We can compare the obtained clustering with the supervised classification of the Iris samples.

```
In [78]: y[::8]
Out[78]: array([0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2])
```

There is a good correlation between the two, but the clustering output has assigned different integer values to the groups than what was used in the target vector in the supervised classification. To compare the two arrays with metrics such as the `confusion_matrix` function, we first need to rename the elements so that the same integer values are used for the same group. We can do this operation with NumPy array manipulations.

```
In [79]: idx_0, idx_1, idx_2 = (np.where(y_pred == n) for n in range(3))
In [80]: y_pred[idx_0], y_pred[idx_1], y_pred[idx_2] = 2, 0, 1
In [81]: y_pred[::8]
Out[81]: array([0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2],
               dtype=int32)
```

Now that we represent the corresponding groups with the same integers, we can summarize the overlaps between the supervised and unsupervised classification of the Iris samples using the `confusion_matrix` function.

```
In [82]: metrics.confusion_matrix(y, y_pred)
Out[82]: array([[50,  0,  0],
                [ 0, 48,  2],
                [ 0, 14, 36]])
```

This confusion matrix indicates that the clustering algorithm correctly identified all samples corresponding to the first species as a group of its own. But due to the overlapping samples in the second and third groups, those could not be resolved entirely as different clusters. For example, two elements from group 1 were assigned to group 2, and 14 elements from group 2 were assigned to group 1.

The clustering result can also be visualized, for example, by plotting scatter plots for each pair of features, as we do in the following. We loop over each pair of features and each cluster and plot a scatter graph for each cluster using different colors (orange, blue, and green in Figure 15-8), and we also draw a red square around each sample for which the clustering does not agree with the supervised classification. The result is shown in Figure 15-8.

```
In [83]: N = X.shape[1]
    ...: fig, axes = plt.subplots(N, N, figsize=(12, 12),
    ...:                          sharex=True, sharey=True)
    ...: colors = ["coral", "blue", "green"]
    ...: markers = ["^", "v", "o"]
    ...: for m in range(N):
    ...:     for n in range(N):
    ...:         for p in range(n_clusters):
    ...:             mask = y_pred == p
    ...:             axes[m, n].scatter(X[:, m][mask], X[:, n][mask], s=30,
    ...:                                marker=markers[p], color=colors[p],
    ...:                                alpha=0.25)
    ...:         for idx in np.where(y != y_pred):
    ...:             axes[m, n].scatter(X[idx, m], X[idx, n], s=30,
    ...:                                marker="s", edgecolor="red",
    ...:                                facecolor=(1,1,1,0))
    ...:     axes[N-1, m].set_xlabel(iris.feature_names[m], fontsize=16)
    ...:     axes[m, 0].set_ylabel(iris.feature_names[m], fontsize=16)
```

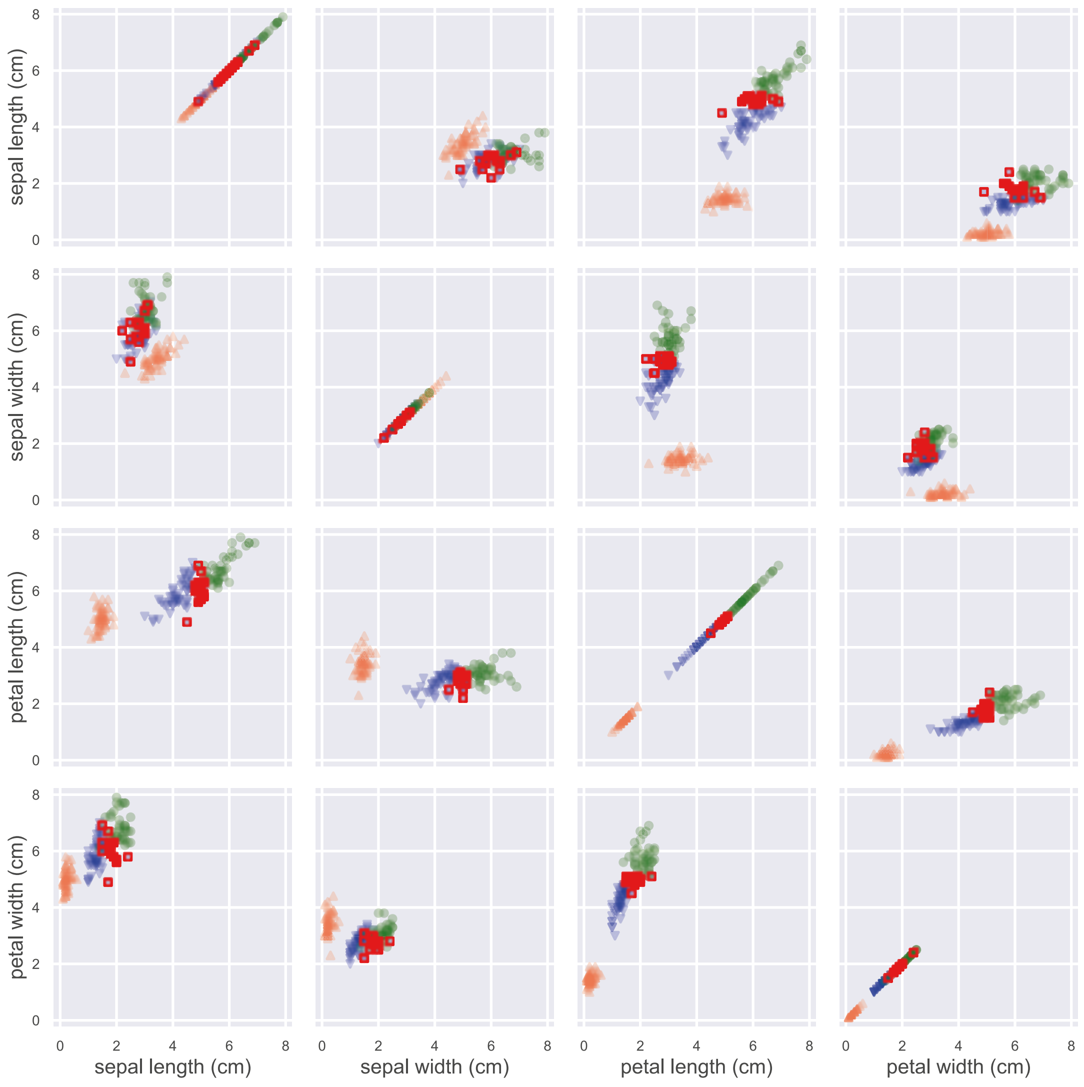

Figure 15-8. *The result of clustering, using the K-means algorithm, of the Iris dataset features*

The Iris samples' clustering result in Figure 15-8 shows that the clustering does remarkably well in recognizing which samples belong to distinct groups. Of course, because of the overlap in the features for classes shown in blue (dark gray) and green (medium gray) in the graph, we cannot expect that any unsupervised clustering algorithm can fully resolve the various groups in the dataset, and some deviation from the supervised response variable is therefore expected.

Summary

This chapter introduced machine learning using Python, beginning with a brief review and summary of the subject and its terminology. The Python library scikit-learn was applied to three different types of problems representing fundamental machine learning topics. We revisited regression from the point of view of machine learning, classification, and clustering. The first two topics are examples of supervised machine learning, while the clustering method is an example of unsupervised machine learning. Beyond what we have been able to cover here, there are many more methods and problem domains covered by machine learning. For example, an essential part of machine learning that we have not touched upon in this brief introduction is text-based problems. The scikit-learn contains an extensive module (`sklearn.text`) with tools and methods for processing text-based problems, and the Natural Language Toolkit (`www.nltk.org`) is a powerful platform for working with and processing data in the form of human language text. Image processing and computer vision are prominent problem domains in machine learning, which, for example, can be treated with OpenCV (`http://opencv.org`) and its Python bindings. Other examples of significant topics in machine learning are neural networks and deep learning, which have received much attention in recent years. Readers interested in such methods should explore the TensorFlow (`www.tensorflow.org`) and the Keras libraries (`https://keras.io`).

Further Reading

Machine learning is a part of the computer science field of artificial intelligence, a broad field with numerous techniques, methods, and applications. This chapter has only shown examples of a few basic machine-learning methods, which can be useful in many practical applications. For a more thorough introduction to machine learning, see *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* by T. Hastie (Springer, 2013), and for introductions to machine learning specific to the Python environment, see *Learning scikit-learn: Machine Learning in Python* by R. Garreta (Packt, 2013), *Mastering Machine Learning With scikit-learn* by G. Hackeling (Packt, 2014), and *Building Machine Learning Systems with Python* by L. Pedro Coelho (Packt, 2015).